4 Large Scale Learning with String Kernels

Sören Sonnenburg Soeren.Sonnenburg@first.fraunhofer.de
Gunnar Rätsch Gunnar.Raetsch@tuebingen.mpg.de
Konrad Rieck Konrad.Rieck@first.fraunhofer.de

In genomic sequence analysis tasks like splice site recognition or promoter identification, large amounts of training sequences are available, and indeed needed to achieve sufficiently high classification performances. In this chapter we study string kernels that can be computed in linear time w.r.t. the length of the input sequences. In particular the recently proposed Spectrum kernel*, the* Weighted Degree kernel *(WD) and the* Weighted Degree kernel with shifts*, which have been successfully used for various sequence analysis tasks. We discuss extensions using data structures such as* tries *and* suffix trees *as well as modifications of a SVM chunking algorithm in order to significantly accelerate SVM training and their evaluation on test sequences. Our simulations using the WD kernel and Spectrum kernel show that large scale SVM training can be accelerated by factors of 7 and 60 times, respectively, while requiring considerably less memory. We demonstrate that these algorithms can be effectively parallelized for further acceleration. Our method allows us to train SVMs on sets as large as 10 million sequences and solve* Multiple Kernel Learning *problems with 1 million sequences. Moreover, using these techniques the evaluation on new sequences is often several thousand times faster, allowing us to apply the classifiers on genome-sized data sets with seven billion test examples. We finally demonstrate how the proposed data structures can be used to understand the SVM classifiers decision function. All presented algorithms are implemented in our Machine Learning toolbox* SHOGUN [1]*.*

4.1 Introduction

Kernel-based methods such as Support Vector Machines (SVMs) have proven to be powerful for a wide range of different data analysis problems, in particular for the analysis of texts and biological sequences. In analysis tasks like *email* or *web*

1. `http://www.fml.tuebingen.mpg.de/raetsch/projects/shogun`.

spam detection (Webb et al., 2006), *splice site recognition* (Rätsch and Sonnenburg, 2004) or *promoter identification* (Sonnenburg et al., 2006c), large amounts of training sequences are available and seemingly required to achieve sufficiently high classification performances. However, training SVMs becomes prohibitively computationally expensive when using genome-sized training samples.

In this work we review and develop string kernels that are particularly well suited for sequence analysis tasks (Section 4.2). We study kernels that consider the occurrence of sub-sequences in the sequence up to a certain length for two typical analysis problems: the analysis of the whole content of the sequences (e.g. web spam classification) and the content relative to a biological signal of interest (e.g. splice sites). We are especially interested in variants of the *Spectrum kernel* (Leslie et al., 2002) and the *Weighted Degree kernel* (WD) (Rätsch and Sonnenburg, 2004), where the latter considers sequences of constant length and uses position dependent information which the former does not.

We will discuss strategies to efficiently compute linear combinations of sequence kernel elements which are frequently used during SVM training and testing. They exploit that the normal vector of the hyperplane in feature space is extremely sparse and build up appropriate index data structures allowing efficient operations on the non-zero elements (Section 4.3). Moreover, we outline algorithms taking advantage of the properties of such kernels in order to efficiently compute the optimal SVM solution (Section 4.4).

Our benchmark experiments in Section 4.5 show that we can significantly speed up the training phase (60× faster for spectrum kernel and ≈ 7× faster for WD kernel) and testing phase (often several thousand times faster). Additionally, during training the algorithms do not require much working memory, as the data structures make memory-demanding kernel caching unnecessary (Sonnenburg et al., 2006b). In Section 4.6 we discuss other applications of the presented ideas. We show how one can solve *Multiple Kernel Learning* (Sonnenburg et al., 2006b) problems with a million examples and how the data structures can be used to obtain representations comprehensible to humans of resulting SVM classifiers.

4.2 String Kernels

Given two strings $\mathbf{x}$ and $\mathbf{x}'$, there is no obvious answer to the question: How similar are $\mathbf{x}$ and $\mathbf{x}'$? In contrast to vectors in $\mathbb{R}^d$ where a quantity inverse to $\|\mathbf{x} - \mathbf{x}'\|$ can be used, similarity of strings can be expressed in a variety of ways – each accounting and emphasizing different features and aspects.

Let us start by defining a few terms: A string (or sequence) $\mathbf{x}$ is defined as $\mathbf{x} \in \Sigma^*$, where Σ^* is the Kleene closure over all symbols from the finite alphabet Σ. The length of the string $\mathbf{x}$ is given as $l_{\mathbf{x}} := |\mathbf{x}|$. Using these definitions we can define similarity measures for use with kernel machines on strings, the so-called *string kernels*. In general there are two major types of string kernels, first the ones that are directly defined on strings and second kernels that are defined on generative models

(like hidden Markov models, e.g. Jaakkola et al., 2000; Tsuda et al., 2002a,b), or by using appropriately defined scores (for instance alignment scores; e.g. Liao and Noble, 2002; Vert et al., 2004).

The following section will cover only string kernels of the first type, such as the Bag-of-Words (Salton, 1979; Joachims, 1998), n-gram (Damashek, 1995; Joachims, 1999a), Locality Improved (Zien et al., 2000), Spectrum (Leslie et al., 2002), Weighted Degree kernel (Rätsch and Sonnenburg, 2004) and Weighted Degree kernel *with shifts* (Rätsch et al., 2005). For additional work that is not directly covered[2] by this work the reader is refered to (Haussler, 1999; Lodhi et al., 2002; Leslie et al., 2003a; Leslie and Kuang, 2004; Schölkopf et al., 2004; Cortes et al., 2004).

4.2.1 Bag-of-Words and n-gram Kernel

In information retrieval a classic way to characterize a text document is to represent the text by the words it contains – the *bag of words* (Salton, 1979). The text document is split at word boundaries into contained words using a set of delimiter symbols, such as space, comma and period. Note that in this representation the ordering of words (e.g. in a sentence) will not be taken into account. The feature space $\mathcal{F}$ consists of all possible words and a document $\mathbf{x}$ is mapped to a sparse vector $\Phi(\mathbf{x}) \in \mathcal{F}$, so that $\Phi_i(\mathbf{x}) = 1$ if the word represented by index i is contained in the document. Further alternatives for mapping $\mathbf{x}$ into a feature space $\mathcal{F}$ correspond to associating $\Phi_i(\mathbf{x})$ with frequencies or counts of contained words. The bag-of-words kernel is then computed as the inner product in $\mathcal{F}$

$$k(\mathbf{x}, \mathbf{x}') = \langle \Phi(\mathbf{x}), \Phi(\mathbf{x}') \rangle = \sum_{i \in \text{words}} \Phi_i(\mathbf{x}) \Phi_i(\mathbf{x}') \tag{4.1}$$

which – in practice – boils down to counting the number of words common to both documents and can thus be computed very efficiently.

Another common approach is to characterize a document by contained n-grams – substrings of n consecutive characters including word boundaries – where n is fixed beforehand (Suen, 1979; Damashek, 1995). The corresponding feature space $\mathcal{F}$ is spanned by all possible strings of length n. Here no dependencies other than the consecutive n characters are taken into account, which however might contain more than one word. The kernel is computed as in (4.1). Note that the n-gram kernel can cope with mismatches, as for instance a single mismatch only affects n neighboring n-grams, while keeping further surrounding ones intact.

---

2. Though the `linadd` optimization trick presented here is - in some cases - also applicable.

4.2.2 The Spectrum Kernel

The spectrum kernel (Leslie et al., 2002) implements the n-gram kernel in the context of biological sequence analysis. The idea is to count how often a d-mer (bioinformatics terminology for d-gram, a contiguous string of length d) is contained in the sequences $\mathbf{x}$ and $\mathbf{x}'$. Summing up the product of these counts for every possible d-mer (note that there are exponentially many) gives rise to the kernel value which formally is defined as follows: Let Σ be an alphabet and $\boldsymbol{u} \in \Sigma^d$ a d-mer and $\#\boldsymbol{u}(\mathbf{x})$ the number of occurrences of $\boldsymbol{u}$ in $\mathbf{x}$. Then the spectrum kernel is defined as:

$$k(\mathbf{x}, \mathbf{x}') = \sum_{\boldsymbol{u} \in \Sigma^d} \#\boldsymbol{u}(\mathbf{x}) \#\boldsymbol{u}(\mathbf{x}') \tag{4.2}$$

Note that spectrum-like kernels cannot extract any positional information from the sequence which goes beyond the d-mer length. It is well suited for describing the content of a sequence but is less suitable for instance for analyzing signals where motifs may appear in a certain order or at specific positions. Also note that spectrum-like kernels are capable of dealing with sequences with varying length.

The spectrum kernel can be efficiently computed in $\mathcal{O}(d(l_{\mathbf{x}} + l_{\mathbf{x}'}))$ using tries (Leslie et al., 2002) and $\mathcal{O}(l_{\mathbf{x}} + l_{\mathbf{x}'})$ using suffix trees (Vishwanathan and Smola, 2003), where $l_{\mathbf{x}}$ and $l_{\mathbf{x}'}$ denote the length of sequence $\mathbf{x}$ and $\mathbf{x}'$. An easier and less complex way to compute the kernel for two sequences $\mathbf{x}$ and $\mathbf{x}'$ is to separately extract and sort the $(l_{\mathbf{x}} + l_{\mathbf{x}'})$ d-mers in each sequence, which can be done in a preprocessing step. Then one iterates over all d-mers of sequences $\mathbf{x}$ and $\mathbf{x}'$ simultaneously, counts which d-mers appear in both sequences and finally sums up the product of their counts. For small alphabets and d-gram lengths individual d-mers can be stored in fixed-size variables, e.g. DNA d-mers of length $d \leq 16$ can be efficiently represented as a 32-bit integer values. The ability to store d-mers in fixed-bit variables or even CPU registers greatly improves performance, as only a single CPU instruction is necessary to compare or index a d-mer. The computational complexity of the kernel computation is $\mathcal{O}(l_{\mathbf{x}} + l_{\mathbf{x}'})$ omitting the preprocessing step.

4.2.3 The Weighted Degree Kernel

The so-called *weighted degree* (WD) kernel (Rätsch and Sonnenburg, 2004) efficiently computes similarities between sequences while taking positional information of multiple k-mers into account. The main idea of the WD kernel is to count the (exact) co-occurrences of k-mers at corresponding positions in the two sequences to be compared. The *WD kernel of order d* compares two sequences $\mathbf{x}$ and $\mathbf{x}'$ of equal length l by summing all contributions of k-mer matches of lengths $k \in \{1, \ldots, d\}$, weighted by coefficients β_k:

$$k(\mathbf{x}, \mathbf{x}') = \sum_{k=1}^{d} \beta_k \sum_{i=1}^{l-k+1} \mathbf{I}(\boldsymbol{u}_{k,i}(\mathbf{x}) = \boldsymbol{u}_{k,i}(\mathbf{x}')). \tag{4.3}$$

Here, $\boldsymbol{u}_{k,i}(\mathbf{x})$ is the string of length k starting at position i of the sequence $\mathbf{x}$ and $\mathbf{I}(\cdot)$ is the indicator function which evaluates to 1 when its argument is *true* and to 0 otherwise. For the weighting coefficients, Rätsch and Sonnenburg (2004) proposed to use $\beta_k = 2\frac{d-k+1}{d(d+1)}$. Matching substrings are thus rewarded with a score depending on the length of the substring. Note that although in our case $\beta_{k+1} < \beta_k$, longer matches nevertheless contribute more strongly than shorter ones: this is due to the fact that each long match implies several short matches, adding to the value of (4.3). Exploiting this knowledge allows for a more intuitive $\mathcal{O}(l)$ reformulation of the kernel using "block-weights" as has been done by Sonnenburg et al. (2005b) and is displayed in Figure 4.1.

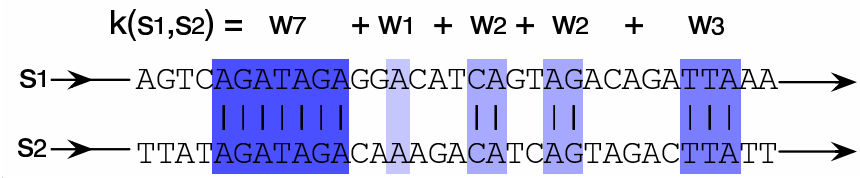


Figure 4.1: Given two sequences $\mathbf{x}_1$ and $\mathbf{x}_2$ of equal length, the kernel consists of a weighted sum to which each match in the sequences makes a contribution w_B depending on its length B, where longer matches contribute more significantly.

Note that the WD kernel can be understood as a spectrum kernel where the k-mers starting at different positions are treated independently of each other. Additionally, the WD kernel considers *substrings of length up to d*. Hence, the feature space for each position has $\sum_{k=1}^{d} |\Sigma|^k = \frac{|\Sigma|^{d+1}-1}{|\Sigma|-1} - 1$ dimensions and is additionally duplicated l times (leading to $\mathcal{O}(l|\Sigma|^d)$ dimensions). However, the computational complexity of the original WD kernel is in the worst case $\mathcal{O}(dl)$ as can be directly seen from (4.3).

4.2.4 The Weighted Degree Kernel *with shifts*

The recognition of matching blocks using the WD kernel strongly depends on the position of the sub-sequence and does not tolerate any positional variation. For instance, if a consecutive block in one sequence is shifted by only one position, the WD kernel fails to discover similar blocks and returns a lower similarity score. Depending on the application in mind, this problem might lead to suboptimal results. Hence, Rätsch et al. (2005) suggested the WD kernel *with shifts* (WDS), which shifts the two sequences against each other in order to tolerate a small positional variations of sequence motifs. Conceptually, it is a mixture between the

WD and the spectrum kernel. It is defined as

$$k(\mathbf{x},\mathbf{x}') = \sum_{k=1}^{d} \beta_k \sum_{i=1}^{l-k+1} \sum_{\substack{s=0 \\ s+i\leq l}}^{S} \delta_s \, \mu_{k,i,s,\mathbf{x},\mathbf{x}'}, \tag{4.4}$$

$$\mu_{k,i,s,\mathbf{x},\mathbf{x}'} = \mathbf{I}(\boldsymbol{u}_{k,i+s}(\mathbf{x}) = \boldsymbol{u}_{k,i}(\mathbf{x}')) + \mathbf{I}(\boldsymbol{u}_{k,i}(\mathbf{x}) = \boldsymbol{u}_{k,i+s}(\mathbf{x}')),$$

where $\beta_k = 2(d-k+1)/(d(d+1))$, $\delta_s = 1/(2(s+1))$ and $\boldsymbol{u}_{k,i}(\mathbf{x})$ is the subsequence of $\mathbf{x}$ of length k that starts at position i. The idea is to count the matches between two sequences $\mathbf{x}$ and $\mathbf{x}'$ between the words $\boldsymbol{u}_{k,i}(\mathbf{x})$ and $\boldsymbol{u}_{k,i}(\mathbf{x}')$ where $\boldsymbol{u}_{k,i}(\mathbf{x}) = x_i x_{i+1} \ldots x_{i+k-1}$ for all i and $1 \leq k \leq d$. The parameter d denotes the maximal length of the words to be compared, and S is the maximum distance by

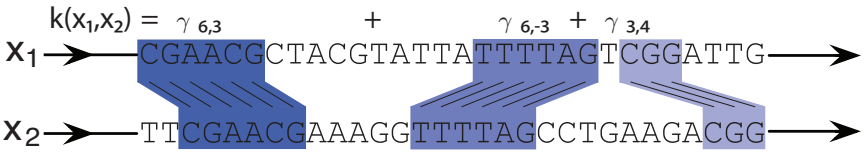


Figure 4.2: Given two sequences $\mathbf{x}_1$ and $\mathbf{x}_2$ of equal length, the WD kernel with shift consists of a weighted sum to which each match in the sequences makes a contribution $\gamma_{k,p}$ depending on its length k and relative position p, where long matches at the same position contribute most significantly. The γ's can be computed from the β's and δ's in (4.4). The spectrum kernel is based on a similar idea, but it only considers substrings of a fixed length and the contributions are independent of the relative positions of the matches to each other.

which a sequence is shifted. Note that this kernel is computationally proportional to the maximum shift l and thus much more demanding requiring $\mathcal{O}(lS)$. See Figure 4.2 and (Rätsch et al., 2005) for a further discussion.

4.2.5 Summary

The string kernels revisited in this section are based on counting substrings of certain lengths. While the computational complexity is linear in the length of the input sequences, there are some significant differences: the spectrum kernel requires $\mathcal{O}(l_{\mathbf{x}}+l_{\mathbf{x}'})$, the WD kernel $\mathcal{O}(l)$ and the WDS kernel is most demanding with $\mathcal{O}(lS)$. As there is no obvious way to further speed up single kernel computations, one can try to exploit the inherent structure of learning algorithms using string kernels. One can particularly benefit from the fact that these algorithms often need to compute linear combinations of kernel elements during training and testing, which can be significantly sped up by exploiting the sparsity of the representation in feature space. Before going into algorithmic details on how one can make use of the sparse feature space to accelerate SVM training and testing in Section 4.4, we will discuss

methods to represent sparse feature maps in the next section (Section 4.3) as the appropriate data representation is crucial for computational efficiency.

4.3 Sparse Feature Maps

The string kernels introduced in the previous section share two important properties: (a) the mapping Φ is explicit, so that elements in the feature space $\mathcal{F}$ can be accessed directly, and (b) mapped examples $\Phi(\mathbf{x})$ are very sparse in comparison to the huge dimensionality of $\mathcal{F}$. In the following sections we illustrate how these properties can be exploited to efficiently store and compute sparse feature vectors.

4.3.1 Efficient Storage of Sparse Weights

The considered string kernels correspond to very large feature spaces, for instance DNA d-mers of order ten span a feature space of over 1 million dimensions. However, most dimensions in the feature space are always zero since only a few of the many different d-mers actually appear in the training sequences, and furthermore a sequence $\mathbf{x}$ can only comprise at most $l_{\mathbf{x}}$ unique d-mers. In this section we briefly discuss four efficient *data structures* for sparse representation of sequences supporting the basic operations: `clear`, `add` and `lookup`. We assume that the elements of a sparse vector $\boldsymbol{v}$ are indexed by some index set $\mathcal{U}$ (for sequences, e.g. $\mathcal{U} = \Sigma^d$). The first operation `clear` sets $\boldsymbol{v}$ to zero. The `add` operation increases either the weight of a dimension of $\boldsymbol{v}$ for an element $\boldsymbol{u} \in \mathcal{U}$ by some amount or increases a set of weights in $\boldsymbol{v}$ corresponding to all d-mers present in a given sequence $\mathbf{x}$. Similar to `add`, the `lookup` operation either requests the value of a particular component $v_{\boldsymbol{u}}$ of $\boldsymbol{v}$ or returns a set of values matching all d-mers in a provided sequence $\mathbf{x}$. The latter two operations need to be performed as quickly as possible.

4.3.1.1 Explicit Map

If the dimensionality of the feature space is small enough, then one might consider keeping the whole vector $\boldsymbol{v}$ in memory and to perform direct operations on its elements. In this case each `add` or `lookup` operation on single elements takes $\mathcal{O}(1)$ time.[3] The approach, however, has expensive memory requirements ($\mathcal{O}(|\Sigma|^d)$), but is highly efficient and best suited for instance for the spectrum kernel on DNA sequences with $d \leq 14$ and on protein sequences with $d \leq 6$.

3. More precisely, it is $\log d$, but for small enough d (which we have to assume anyway) the computational effort is exactly one memory access.

4.3.1.2 Sorted Arrays and Hash Tables

More memory efficient but computationally more expensive are sorted arrays of index-value pairs $(\boldsymbol{u}, v_{\boldsymbol{u}})$. Assuming $l_{\mathbf{x}}$ indices of a sequence $\mathbf{x}$ are given and sorted in advance, one can efficiently `add` or `lookup` up a single $v_{\boldsymbol{u}}$ for a corresponding $\boldsymbol{u}$ by employing a binary search procedure with $\mathcal{O}(\log l_{\mathbf{x}})$ run-time. Given a sequence $\mathbf{x}'$ to look up all contained d-mers at once, one may sort the d-mers of $\mathbf{x}'$ in advance and then simultaneously traverse the arrays of $\mathbf{x}$ and $\mathbf{x}'$ in order to determine which elements appear in $\mathbf{x}'$. This procedure results in $\mathcal{O}(l_{\mathbf{x}} + l_{\mathbf{x}'})$ operations – omitting the sorting of the second array – instead of $\mathcal{O}(l_{\mathbf{x}'} \log l_{\mathbf{x}})$. The approach is well suited for cases where $l_{\mathbf{x}}$ and $l_{\mathbf{x}'}$ are of comparable size, as for instance for computations of single spectrum kernel elements (Leslie et al., 2003b). If $l_{\mathbf{x}} \gg l_{\mathbf{x}'}$, then the binary search procedure should be preferred.

A trade-off between the efficiency of explicit maps and the low memory requirements of sorted arrays can be achieved by storing the index-value pairs $(\boldsymbol{u}, v_{\boldsymbol{u}})$ in a hash table, where $\boldsymbol{u}$ is hashed to a bin in the hash table containing $v_{\boldsymbol{u}}$ (Rieck et al., 2006a). Both operations `add` and `lookup` for $\boldsymbol{u} \in \Sigma^d$ can be carried out in $\mathcal{O}(1)$ time in best-case – however the worst-case run-time is $\mathcal{O}(\log l_{\mathbf{x}})$, if all $\boldsymbol{u}$ of a sequence $\mathbf{x}$ are mapped to the same bin. The two opposed runtime bounds suggest that the hash table size has to be chosen very carefully in advance and also strongly depends on the lengths of the considered sequences, which makes the sorted array approach more practicable in terms of run-time requirements.

4.3.1.3 Tries

Another way of organizing the non-zero elements are *tries* (Fredkin, 1960; Knuth, 1973): The idea is to use a tree with at most $|\Sigma|$ siblings of depth d. The leaves store a single value: the element $v_{\boldsymbol{u}}$, where $\boldsymbol{u} \in \Sigma^d$ is a d-mer and the path to the leaf corresponds to $\boldsymbol{u}$.

To `add` an element to a trie one needs $\mathcal{O}(d)$ in order to create the necessary nodes on the way from the root to a leave. Similar to the `add` operation, a `lookup` takes $\mathcal{O}(d)$ time in the worst-case, however, with growing d the probability for an arbitrary $\boldsymbol{u}$ to be present in a trie decreases exponentially, so that a logarithmic run-time of $\mathcal{O}(\log_{|\Sigma|} d)$ can be expected for large d. Note that the worst-case computational complexity of both operations is independent of the number of d-mers/elements stored in the tree.

Tries need considerably more storage than sorted arrays (for instance storing edges in nodes usually requires using hash tables or balanced tree maps), however, tries are useful for the previously discussed WD kernel. Here we not only have to lookup one substring $\boldsymbol{u} \in \Sigma^d$, but also all prefixes of $\boldsymbol{u}$. For sorted arrays this amounts to d separate `lookup` operations, while for tries all prefixes of $\boldsymbol{u}$ are already known when the bottom of the trie is reached. In this case the trie has to store aggregated weights in internal nodes (Sonnenburg et al., 2006b; Rieck et al., 2006a). This is illustrated for the WD kernel in Figure 4.3.

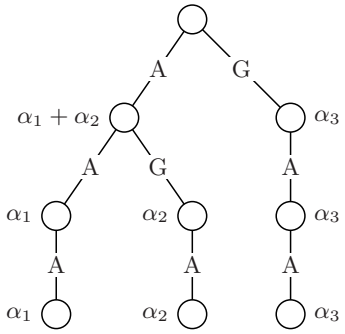
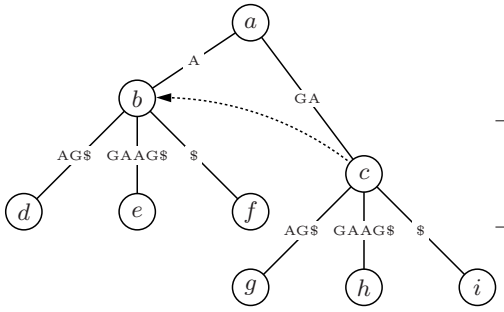


Figure 4.3: Trie containing the 3-mers AAA, AGA, GAA with weights α_1, α_2, α_3. Additionally the figure displays resulting weights at inner nodes.

4.3.1.4 Suffix Trees and Matching Statistics

A fourth alternative for efficient storage of sparse weights and string kernel computation builds on two data structures: suffix trees and matching statistics (Vishwanathan and Smola, 2003). A *suffix tree* $S_{\mathbf{x}}$ is a compact representation of a trie, which stores all suffixes of a sequence $\mathbf{x}$ in $\mathcal{O}(l_{\mathbf{x}})$ space and allows efficient retrieval of arbitrary sub-sequences of $\mathbf{x}$ (Gusfield, 1997). A *matching statistic* $M_{\mathbf{x}'}$ for a suffix tree $S_{\mathbf{x}}$ is defined by two vectors v and c of length $l_{\mathbf{x}'}$, where v_i reflects the length of the longest substring of $\mathbf{x}$ matching $\mathbf{x}'$ at position i and c_i is a corresponding node in $S_{\mathbf{x}}$ (Chang and Lawler, 1994). As an example Figure 4.4 shows a suffix tree $S_{\mathbf{x}}$ and a matching statistic $M_{\mathbf{x}'}$ for the sequences $\mathbf{x} =$ GAGAAG and $\mathbf{x}' =$ GAACG.



(a) Suffix tree $S_{\mathbf{x}}$

$\mathbf{x}'$	G	A	A	C	G
v_i	3	2	1	0	1
c_i	g	d	b	a	c

(b) Matching statistic $M_{\mathbf{x}'}$

Figure 4.4: Suffix tree $S_{\mathbf{x}}$ for the sequence $\mathbf{x} =$ GAGAAG and matching statistic $M_{\mathbf{x}'}$ for $\mathbf{x}' =$ GAACG matched against $S_{\mathbf{x}}$. A sentinel symbol \$ has been added to $\mathbf{x}$, s.t. that all leaves correspond to suffixes.

By traversing $S_{\mathbf{x}}$ and looping over $M_{\mathbf{x}'}$ in parallel a variety of string kernels $k(\mathbf{x}, \mathbf{x}')$ – including the spectrum and WD kernel – can be computed using $\mathcal{O}(l_{\mathbf{x}}+l_{\mathbf{x}'})$ run-time. A detailed discussion of the corresponding algorithms, weighting schemes

and extensions is given in (Vishwanathan and Smola, 2004; Rieck et al., 2006b). The approach can be further extended to support the operations `clear`, `add` and `lookup`. In contrast to sorted arrays and tries, these operations are favorably performed on the domain of *sequences* instead of single d-mers to ensure linear run-time. Starting with an empty suffix tree S obtained using `clear`, the `add` operation is realized by appending sequences and implicitly contained d-mers to S using an online construction algorithm, (e.g. Ukkonen, 1995). In order to avoid matches over multiple sequences, each sequence $\mathbf{x}_i$ is delimited by a sentinel symbol $\$_i \notin \Sigma$. Given S, the `lookup` operation for a sequence $\mathbf{x}'$ is performed by calculating $M_{\mathbf{x}'}$, so that a kernel computation can be carried out in $\mathcal{O}(l_{\mathbf{x}'})$ run-time using S and $M_{\mathbf{x}'}$.

In practice however, suffix trees introduce a crucial overhead in storage space due to the high complexity of the data structure, which makes memory preserving data structures such as sorted arrays more attractive, especially on small alphabets. Recently an alternative, suffix arrays, have been proposed to reduce the memory requirements, still n input symbols results in at least $19n$ bytes of allocated memory independent of the considered alphabet (Teo and Vishwanathan, 2006)

4.3.1.5 Summary

Table 4.1 coarsely summarizes the worst-case run-times for multiple calls of `clear`, `add` and `lookup` using the previously introduced data structures. From the provided run-time bounds, it is obvious that the explicit map representation is favorable if the considered alphabet Σ and order d is sufficiently small – for instance as in several application of DNA analysis where $|\Sigma| = 4$ and $d \leq 6$. For larger alphabets and higher d, the sorted array approach is more attractive in practice. Other than tries and suffix trees, the sorted array approach is much easier to implement and its memory requirements are easier to estimate. If either Σ or d can get arbitrarily large, suffix trees are the data structure of choice as they operate linear in sequence lengths independent of Σ and d, however, as mentioned earlier there is a large overhead in storage space due to the complexity of the suffix tree structure. The trie-based approach may not seem suitable for large scale learning in comparison to the other methods, but the per-node augmentation of tries with additional values such as aggregated weights shown in Figure 4.3 can drastically speed up computation of complex string kernels such as the WD kernel, which can not efficiently be mapped to other approaches.

	Explicit map	Sorted arrays	Tries	Suffix trees
`clear` of $\boldsymbol{v}$	$\mathcal{O}(\lvert\Sigma\rvert^d)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`add` of all $\boldsymbol{u}$ from $\mathbf{x}$ to $\boldsymbol{v}$	$\mathcal{O}(l_{\mathbf{x}})$	$\mathcal{O}(l_{\mathbf{x}} \log l_{\mathbf{x}})$	$\mathcal{O}(l_{\mathbf{x}} d)$	$\mathcal{O}(l_{\mathbf{x}})$
`lookup` of all $\boldsymbol{u}$ from $\mathbf{x}'$ in $\boldsymbol{v}$	$\mathcal{O}(l_{\mathbf{x}'})$	$\mathcal{O}(l_{\mathbf{x}} + l_{\mathbf{x}'})$	$\mathcal{O}(l_{\mathbf{x}'} d)$	$\mathcal{O}(l_{\mathbf{x}'})$

Table 4.1: Comparison of worst-case run-times for multiple calls of `clear`, `add` and `lookup` on a sparse vector $\boldsymbol{v}$ using explicit maps, sorted arrays, tries and suffix trees.

4.4 Speeding up SVM Training and Testing

As it is not feasible to use standard optimization toolboxes for solving large scale SVM training problems, decomposition techniques are frequently used in practice. Most chunking algorithms work by first selecting Q variables (working set $W \subseteq \{1, \ldots, N\}$, $Q := |W|$) based on the current solution and then solve the reduced problem with respect to the working set variables. These two steps are repeated until some optimality conditions are satisfied as displayed in Algorithm 4.1.

Algorithm 4.1 A SVM Chunking Algorithm (see e.g. Joachims, 1998).

while optimality conditions are violated **do**
  select Q variables for the working set.
  solve reduced problem on the working set.
end while

For selecting the working set and checking the termination criteria in each iteration, the vector $\mathtt{g}$ with $g_i = \sum_{j=1}^{N} \alpha_j y_j k(x_i, x_j)$, $i = 1, \ldots, N$, is usually needed. Without further assumptions, computing $\mathtt{g}$ from scratch in every iteration requires $\mathcal{O}(N^2)$ kernel computations. To avoid re-computation of $\mathtt{g}$ one typically starts with $\mathtt{g} = \mathbf{0}$ and only computes updates of $\mathtt{g}$ on the working set W

$$g_i \leftarrow g_i^{old} + \sum_{j \in W} (\alpha_j - \alpha_j^{old}) y_j k(x_i, x_j) \tag{4.5}$$

for all $i = 1, \ldots, N$. As a result the effort decreases to $\mathcal{O}(QN)$ kernel computations, which can be further accelerated by using kernel caching (Joachims, 1998). However, kernel caching is not efficient enough for large scale problems[4] and thus most time is spent computing kernel rows for the updates of $\mathtt{g}$ on the working set W. Note however that this update as well as computing the Q kernel rows can be easily parallelized (see Section 4.5.2).

Exploiting $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle$ and $\mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \Phi(\mathbf{x}_i)$ we can rewrite the update rule as

$$g_i \leftarrow g_i^{old} + \sum_{j \in W} (\alpha_j - \alpha_j^{old}) y_j \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle = g_i^{old} + \langle \mathbf{w}^W, \Phi(\mathbf{x}_i) \rangle, \tag{4.6}$$

where $\mathbf{w}^W = \sum_{j \in W} (\alpha_j - \alpha_j^{old}) y_j \Phi(\mathbf{x}_j)$ is the normal vector on the working set. If the kernel feature map can be computed explicitly and is sparse (as discussed before), then computing the update in (4.6) can be accelerated. One only needs to compute and store $\mathbf{w}^W$ (using the `clear` and $\sum_{q \in W} |\{\Phi_j(\mathbf{x}_q) \neq 0\}|$ `add` operations)

4. For instance when using a million examples one can only fit 268 rows into 1 GB. Moreover, caching 268 rows is insufficient when for instance having many thousands of active variables.

and performing the scalar product $\langle \mathbf{w}^W, \Phi(\mathbf{x}_i)\rangle$ (using $|\{\Phi_j(\mathbf{x}_i) \neq 0\}|$ `lookup` operations).

Depending on the kernel, the way the sparse vectors are stored (cf. Section 4.3.1) and on the sparseness of the feature vectors, the speedup can be quite drastic. For instance, a single WD kernel computation (as in (4.3)) requires $\mathcal{O}(dl)$ operations. Hence, computing (4.5) N times requires $\mathrm{O}(NQld)$ operations. When using tries for computing (4.6), one needs Ql `add` operations (each $\mathcal{O}(d)$) and Nl `lookup` operations (each $\mathcal{O}(d)$). Hence, only $\mathcal{O}(Qld + Nld)$ basic operations are needed in total. When N is large enough it leads to a speedup by a factor of Q. Finally note that kernel caching is no longer required and as Q is small in practice (e.g. $Q = 42$) the resulting trie has rather few leaves and thus only needs little storage.

The pseudo-code of our `linadd` SVM chunking algorithm is given in Algorithm 4.2.

Algorithm 4.2 Outline of the chunking algorithm that exploits the fast computations of linear combinations of kernels (e.g. by tries).

INITIALIZATION
$g_i = 0$, $\alpha_i = 0$ for $i = 1, \ldots, N$
LOOP UNTIL CONVERGENCE
for $t = 1, 2, \ldots$ **do**
  Check optimality conditions and stop if optimal
  select working set W based on $\mathbf{g}$ and $\boldsymbol{\alpha}$, store $\boldsymbol{\alpha}^{old} = \boldsymbol{\alpha}$
  solve reduced problem W and update $\boldsymbol{\alpha}$
  `clear w`
  $\mathbf{w} \leftarrow \mathbf{w} + (\alpha_j - \alpha_j^{old}) y_j \Phi(\mathbf{x}_j)$ for all $j \in W$ (using `add`)
  update $g_i = g_i + \langle \mathbf{w}, \Phi(\mathbf{x}_i)\rangle$ for all $i = 1, \ldots, N$ (using `lookup`)
end for

4.4.1 A Parallel Chunking Algorithm

As still most time is spent in evaluating $g(\mathbf{x})$ for all training examples further speedups are gained when parallelizing the evaluation of $g(\mathbf{x})$. When using the `linadd` algorithm, one first constructs the data structure representing the update vector $\mathbf{w}$ and then performs parallel `lookup` operations using several CPUs (e.g. using shared memory or several copies of the data structure on separate computing nodes). We have implemented this algorithm based on multiple *threads* and gain reasonable speedups (see next section).

Note that this part of the computations is almost ideal to distribute to many CPUs, as only the updated $\boldsymbol{\alpha}$ (or $\mathbf{w}$ depending on the communication costs and size) have to be transferred before each CPU computes a large chunk $I_k \subset \{1, \ldots, N\}$ of

$$h_i^{(k)} = \langle \mathbf{w}, \Phi(\mathbf{x}_i)\rangle, \quad \forall i \in I_k, \quad \forall k = 1, \ldots, N, \text{ where } (I_1 \cup \cdots \cup I_n) = (1, \ldots, N)$$

which is transferred to a master node that finally computes $\mathbf{g} \leftarrow \mathbf{g} + \mathbf{h}$, as illustrated

in Algorithm 4.3.

Algorithm 4.3 Outline of the parallel chunking algorithm that exploits the fast computations of linear combinations of kernels.

Master node
INITIALIZATION
$g_i = 0$, $\alpha_i = 0$ for $i = 1, \ldots, N$
LOOP UNTIL CONVERGENCE
for $t = 1, 2, \ldots$ **do**
  Check optimality conditions and stop if optimal
  select working set W based on $\mathbf{g}$ and $\boldsymbol{\alpha}$, store $\boldsymbol{\alpha}^{old} = \boldsymbol{\alpha}$
  solve reduced problem W and update $\boldsymbol{\alpha}$
  transfer to Slave nodes: $\alpha_j - \alpha_j^{old}$ for all $j \in W$
  fetch from n Slave nodes: $\mathbf{h} = (\mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(n)})$
  update $g_i = g_i + h_i$ for all $i = 1, \ldots, N$
end for
signal convergence to slave nodes

Slave nodes
LOOP UNTIL CONVERGENCE
while not converged **do**
  fetch from Master node $\alpha_j - \alpha_j^{old}$ for all $j \in W$
  `clear w`
  $\mathbf{w} \leftarrow \mathbf{w} + (\alpha_j - \alpha_j^{old}) y_j \Phi(\mathbf{x}_j)$ for all $j \in W$ (using `add`)
  node k computes $h_i^{(k)} = \langle \mathbf{w}, \Phi(\mathbf{x}_i) \rangle$
    for all $i = \lceil (k-1)\frac{N}{n} \rceil + 1, \ldots, \lceil k\frac{N}{n} \rceil$ (using `lookup`)
  transfer to master: $\mathbf{h}^{(k)}$
end while

4.5 Benchmark Experiments

In this section we will perform a benchmark comparing the running times of SVMs using the proposed algorithmic optimizations.

4.5.1 Experimental Setup

To demonstrate the effect of the `linadd` SVM training optimizations (Algorithm 4.2) we applied the standard and the `linadd` algorithm using 1 to 8 CPUs to a *human* splice site data set,[5] comparing it to the original WD and spectrum kernel formulation. The splice data set contains 159,771 true acceptor splice site sequences

5. The splice dataset can be downloaded from `http://www.fml.tuebingen.mpg.de/raetsch/projects/lsmkl`

and 14,868,555 decoys, leading to a total of 15,028,326 sequences each 141 base pairs in length. It was generated following a procedure similar to the one of Sonnenburg et al. (2005a) for *C. elegans*, which only contained 1,026,036 examples. Note that the dataset is very unbalanced as 98.94% of the examples are negatively labeled. For training we selected 500, 1,000, 5,000, 10,000, 30,000, 50,000, 100,000, 200,000, 500,000, 1,000,000, 2,000,000, 5,000,000 and 10,000,000 randomly sub-sampled examples and measured the time needed in SVM training. For classification performance evaluation we always use the same remaining 5,028,326 examples as a test data set. We set the degree parameter to $d = 20$ for the WD kernel and to $d = 8$ for the spectrum kernel fixing the SVM's regularization parameter to $C = 1$. We used tries for the WD kernel and explicit maps with 2^{16} elements for the spectrum kernel as the DNA alphabet requires only two bits to enumerate the four symbols A, C, G, T leading to 16-bit 8-th order words.

Since the spectrum kernel is position independent it is not well suited for the splice site recognition problem that requires knowledge of the position of substring relative to the splice site. We therefore applied this kernel to a *web spam* data set ($d = 4$), where the task is to distinguish web pages that are maliciously tailored to achieve high ranks by search engines – so called web spam – from normal web pages. As negative examples we obtained the Webb Spam Corpus[6] (Webb et al., 2006), which comprises about 350,000 pages of web spam. In order to generate normal data, we selected an initial set of popular web sites (e.g. `cnn.com`, `microsoft.com`, `slashdot.org` and `heise.de`) and recursively followed links up to a depth of 3, resulting in 250,000 downloaded web pages from more than 10,000 web sites. The average length of the web pages is 20 Kb with a standard deviation of 25 Kb. We then filtered all pages that did not contain the <`html`> tag (case insensitive matching) leading to 300,000 spam and 180,000 normal pages with an average size of 30 Kb per page and a total size of 5 GB. As a sparse mapping we used sorted arrays of 64-bit unsigned integers allowing us to consider up to 8-mers, due to the fact that some of the retrieved web pages are in fact binaries (8-bit alphabet: $0 \dots 255$). We used a random subset of 100,000 examples for training and a separate set of the same size for testing. The total size of the training and test data set is ≈ 4 GB, which results in ≈ 30 GB of memory requirements using sorted arrays of 64-bit variables.

The splice and the web spam data sets are used in all benchmark experiments and SVMs are trained using the SHOGUN machine learning toolbox, [7] which contains a modified version of SVM^{light} (Joachims, 1999a). SVM$^{\text{light}}$'s subproblem size (parameter `qpsize`) and convergence criterion (parameter `epsilon`) were set to $Q = 42$ and $\epsilon_{SVM} = 10^{-5}$. See Table 4.6 for other choices of Q. A kernel cache of 1 GB was used for all kernels except the precomputed kernel and algorithms using the `linadd` extension for which the kernel-cache was disabled. Experiments

6. Available from `http://spamarchive.org/gt/`
7. The toolbox source code is freely available from `http://www.fml.tuebingen.mpg.de/raetsch/projects/shogun`.

were performed on a PC powered by *eight* 2.4GHz AMD Opteron(tm) processors running Linux. We measured the training time for each of the algorithms (single, quad or eight CPU version) and data set sizes.

4.5.2 Benchmarking SVM

Splice Dataset The obtained training times for the different SVM algorithms on the splice dataset are displayed in Table 4.2, 4.3 and in Figure 4.5. First, SVMs were trained using standard SVM$^{\text{light}}$ with the Weighted Degree Kernel precomputed (*WDPre*), the standard WD kernel (*WD1*) and the precomputed (*SpecPre*) and standard spectrum kernel (*Spec*). Then SVMs utilizing the `linadd` extension[8] were trained using the WD (*LinWD*) and spectrum (*LinSpec*) kernel. Finally, SVMs were trained on four and eight CPUs using the parallel version of the `linadd` algorithm (*LinWD4, LinWD8*). *WD4* and *WD8* demonstrate the effect of a simple parallelization strategy, where the computation of kernel rows and updates on the working set are parallelized, which works with *any* kernel.

N	*WDPre*	*WD1*	*WD4*	*WD8*	*LinWD1*	*LinWD4*	*LinWD8*
500	1	1	1	1	1	1	1
1,000	2	1	1	1	3	1	1
5,000	29	7	5	5	16	4	5
10,000	109	19	13	12	35	10	11
30,000	934	110	52	45	136	33	27
50,000	-	448	170	125	254	61	45
100,000	-	1233	472	386	309	101	84
200,000	-	4460	1543	1284	779	220	166
500,000	-	22229	8664	6998	2978	898	611
1,000,000	-	-	-	-	7189	2366	1474
2,000,000	-	-	-	-	-	-	4274
5,000,000	-	-	-	-	-	-	18547
10,000,000	-	-	-	-	-	-	97484

Table 4.2: Speed comparison of the standard single CPU Weighted Degree Kernel algorithm (*WD1*) in SVM$^{\text{light}}$ training, compared to the four (*WD4*) and eight (*WD8*) CPUs parallelized version, the precomputed version (Pre) and the `linadd` extension used in conjunction with the standard WD kernel for 1,4 and 8 CPUs (*LinWD1, LinWD4, LinWD8*) on the splice dataset

.

8. More precisely the $\mathcal{O}(l)$ block formulation of the WD kernel as proposed by Sonnenburg et al. (2005b) was used in all WD-SVM benchmarks (potentially in addition to the `linadd` extension).

N	*SpecPre*	*Spec*	*LinSpec1*	*LinSpec4*	*LinSpec8*
500	1	1	1	1	1
1,000	2	2	1	1	1
5,000	27	38	4	2	3
10,000	104	117	4	5	4
30,000	915	715	33	17	13
50,000	-	1207	38	25	26
100,000	-	3982	127	64	84
200,000	-	14200	419	254	283
500,000	-	181241	3027	1719	1611
1,000,000	-	-	27350	14581	12991

Table 4.3: Speed comparison of the spectrum kernel without (*Spec*) and with `linadd` (*LinSpec1*, *LinSpec4*, *LinSpec8* using 1,4 and 8 processors) on the splice data set. *SpecPre* denotes the precomputed version. The first column shows the sample size N of the data set used in SVM training while the following columns display the time (measured in seconds) needed in the training phase.

The training times obtained when precomputing the kernel matrix (which includes the time needed to precompute the full kernel matrix) is in all cases larger than the times obtained using the standard WD kernel demonstrating the effectiveness of SVM$^{\text{light}}$'s kernel cache. The overhead of constructing a trie on $Q = 42$ examples is visible: only starting from 50,000 examples `linadd` optimization becomes more efficient than the original WD kernel algorithm as the kernel cache cannot hold all kernel elements anymore.[9]

The `linadd` formulation outperforms the original WD kernel by a factor of 7.5 on half a million examples. The picture is similar for the spectrum kernel, here speedups of factor 59.9 on $500,000$ examples are reached which stems from the fact that explicit maps (and not tries as in the WD kernel case) as discussed in Section 4.3.1 were used. This led to a `lookup` cost of $\mathcal{O}(1)$ and a dramatically reduced map construction time. For that reason the parallelization effort benefits the WD kernel more than the spectrum kernel: on half a million examples the parallelization using 4 CPUs (8 CPUs) leads to a speedup of factor 3.3 (4.9) for the WD kernel, but only 1.8 (1.9) for the spectrum kernel. Thus parallelization will help more if the kernel computation is slow. Training with the original WD kernel with a sample size of $500,000$ takes about 6 hours, the `linadd` version still requires about 50 minutes while with the 8 CPU parallel implementation requires about 2 hours and in conjunction with the `linadd` optimization only 20 minutes are needed. Finally, training on 10 million examples takes about 27 hours. Note that this data

9. When single precision 4-byte floating point numbers are used, caching all kernel elements is possible when training with up to 16384 examples.

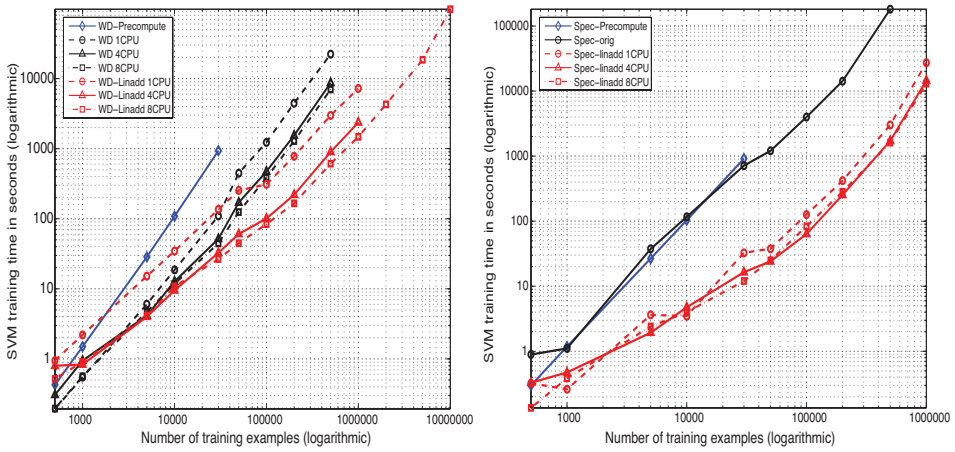


Figure 4.5: Comparison of the running time of the different SVM training algorithms using the weighted degree kernel on the splice data set. Note that as this is a log-log plot small appearing distances are large for larger N and that each slope corresponds to a different exponent. In the left figure the Weighted Degree kernel training times are measured, the right figure displays spectrum kernel training times.

set is already 2.1 GB in size.

N	100	500	1,000	5,000	10,000	20,000	50,000	70,000	100,000
Spec	2	97	201	1977	6039	19063	94012	193327	-
LinSpec	3	255	840	4030	9128	11948	44706	83802	107661
Accuracy	89.59	92.12	93.50	96.36	97.03	97.46	97.83	97.98	98.18
auROC	94.37	97.82	98.41	99.11	99.32	99.43	99.59	99.61	99.64

Table 4.4: Speed and classification accuracy comparison of the spectrum kernel without (*Spec*) and with `linadd` (*LinSpec*) on the web spam data set. The first row shows the sample size N of the data set used in SVM training. The next two rows display the time (measured in seconds) needed in the training phase, followed by the classification accuracy and and area under the Receiver Operator Characteristic Curve (in percent, cf. Section 4.5.3)

Web Spam Dataset Table 4.4 lists measured run-times with and without the `linadd` optimization on the web spam data set introduced in Section 4.5.1, as well as classification accuracy. As before, we used $Q = 42$ as quadratic sub-problem size. A discussion on the sub-problem size for the splice dataset can be found in Section 4.5.4 suggesting that a mid-range $Q = 42$ works best in most

cases. Similarly to the splice data set, the `linadd` optimization yields performance improvements for larger training sets, e.g. for 70,000 training instances of up to a factor of 2.3. The classification accuracy steadily increases with the number of training examples and finally reaches an accuracy of 98.18% and an area under the Receiver Operator Characteristic Curve of 99.64% on 100,000 examples (for a discussion of the performance measures, see Section 4.5.3).

The large alphabet, however, requires utilization of sorted arrays in contrast to the explicit map representation used for the splice data set. Furthermore, the web spam pages are on average 200 times longer than DNA sequences in the splice data set. As a result the speedup achieved through the `linadd` optimization is limited by maintenance of the sorted arrays and, hence, the less effective CPU cache. Note that as web documents have an average size of 30 kb, training on 100,000 examples requires $\approx$ 15 GB of memory just to store the 64-bit variables in sorted arrays.

4.5.3 Classification Performance

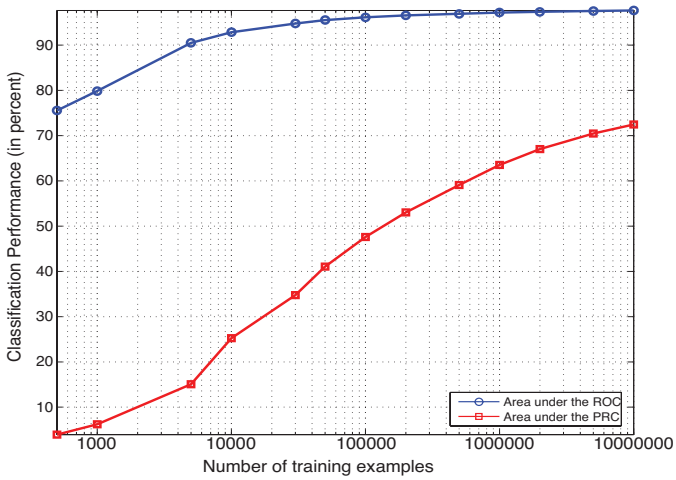


Figure 4.6: Comparison of the classification performance of the Weighted Degree kernel based SVM classifier for different training set sizes. The area under the Receiver Operator Characteristic (ROC) Curve and the area under the Precision Recall Curve (PRC) are displayed (in percent). Note that as this is a very unbalanced dataset, the area under the ROC curve is less meaningful than the area under the PRC.

Figure 4.6 and Table 4.5 show the classification performance[10] in terms of area

10. We omit to show the classification accuracy, as 98.94% of the examples are negatively labeled. Thus, the simplest classifier, predicting -1 for all examples, already achieves

under the Receiver Operator Characteristic (ROC) Curve (Metz, 1978; Fawcett, 2003) and the area under the Precision Recall Curve (PRC) (see e.g. Davis and Goadrich, 2006) of SVMs on the human splice data set for different data set sizes using the WD kernel.

Recall the definition of the ROC and PRC curves: The sensitivity (or recall) is defined as the fraction of correctly classified positive examples among the total number of positive examples, i.e. it equals the true positive rate $TPR = TP/(TP + FN)$. Analogously, the fraction $FPR = FP/(TN + FP)$ of negative examples wrongly classified as positive is called the false positive rate. Plotting FPR against TPR results in the Receiver Operator Characteristic Curve (ROC) (Metz, 1978; Fawcett, 2003). Plotting the true positive rate against the positive predictive value (also precision) $PPV = TP/(FP + TP)$, i.e. the fraction of correct positive predictions among all positively predicted examples, one obtains the Precision Recall Curve (PRC) (see e.g. Davis and Goadrich, 2006). Note that this is a very unbalanced dataset. Hence, the area under the ROC curve is rather meaningless, since this measure is independent of class ratios. The area under the Precision Recall Curve (auPRC) seems a more sensible measure here. It steadily increases as more training examples are used for learning. Thus one should train using all available data to obtain the best results.

N	*auROC*	*auPRC*
500	75.55	3.94
1,000	79.86	6.22
5,000	90.49	15.07
10,000	92.83	25.25
30,000	94.77	34.76
50,000	95.52	41.06
100,000	96.14	47.61
200,000	96.57	53.04
500,000	96.93	59.09
1,000,000	97.19	63.51
2,000,000	97.36	67.04
5,000,000	97.54	70.47
10,000,000	97.67	72.46
10,000,000	96.03*	44.64*

Table 4.5: Comparison of the classification performance of the Weighted Degree kernel based SVM classifier for different training set sizes. The area under the ROC curve (*auROC*) and the area under the Precision Recall Curve (*auPRC*) are displayed (in percent). Larger values are better. An optimal classifier would achieve 100% Note that as this is a very unbalanced dataset the area under the ROC curve is almost meaningless. For comparison, the classification performance achieved using a 4th order Markov chain on 10 million examples is displayed in the last row (marked *; order 4 was chosen based on model selection, where orders 1-8 using several values for the pseudo-count were considered).

98.94% rendering the accuracy measure meaningless.

N	Q 12	32	42	52	72	92	112	132	152
500	2	1	1	2	2	1	2	2	2
1,000	4	3	3	3	3	3	3	3	3
5,000	22	16	16	15	15	15	16	16	17
10,000	51	**35**	**35**	**36**	39	42	43	43	43
30,000	204	**138**	**136**	**139**	148	156	165	175	**132**
50,000	397	**264**	**254**	**266**	**272**	290	303	315	327
100,000	449	**317**	**309**	368	344	374	387	721	752
200,000	1107	**771**	**779**	**848**	**796**	867	1573	940	1670
500,000	4691	**2754**	2978	**2714**	2910	3063	4369	3995	3457
1,000,000	14429	8211	**7189**	8462	8524	9857	9574	8727	9077

Table 4.6: Influence on training time when varying the size of the quadratic program Q in SVM$^{\text{light}}$, when using the `linadd` formulation of the WD kernel. While training times do not vary dramatically one still observes the tendency that with larger sample size a larger Q becomes optimal. The $Q = 42$ column displays the same result as column *LinWD1* in Table 4.2.

4.5.4 Varying the subproblem size Q

As discussed in Section 4.4 and Algorithm 4.2, using the `linadd` algorithm for computing the output for all training examples w.r.t. to some working set can be sped up by a factor of Q (i.e. the size of the quadratic subproblems, termed `qpsize` in SVM$^{\text{light}}$). However, there is a trade-off in choosing Q as solving larger quadratic subproblems is expensive (quadratic to cubic effort). Table 4.6 shows the dependence of the computing time from Q and N. For example choosing $Q = 42$ instead of $Q = 12$ for 1 million examples leads to a speedup of factor 2. Sticking with a mid-range Q (here $Q = 42$) seems to be a good idea for this task. However, a large variance can be observed, as the SVM training time depends to a large extent on which Q variables are selected in each optimization step. For example, on the related *C. elegans* splice data set $Q = 141$ was optimal for large sample sizes, while a mid-range $Q = 71$ led to the overall best performance. Here any $Q > 12$ leads to a similar SVM training time.

4.6 Extensions

In this section we show that the `linadd` extensions are especially helpful for Multiple Kernel Learning (Section 4.6.1). We show that speedups of up to a factor 50 for the WD kernel are possible (using a single CPU). In Section 4.6.2 we discuss methods for improving the interpretability of the classifier. We show that we can extract useful knowledge from the learned decision boundaries by using our data

structures. Moreover, we illustrate that Multiple Kernel Learning can also be helpful for understanding which information is used by the SVM for discrimination. We study several toy and real world examples for illustration.

4.6.1 Large Scale Multiple Kernel Learning

In the Multiple Kernel Learning (MKL) problem for binary classification one is given N data points $(\mathbf{x}_i, y_i)$ ($y_i \in \{\pm 1\}$), where $\mathbf{x}_i$ is translated via K mappings $\Phi_k(\mathbf{x}) \mapsto \mathbb{R}^{D_k}$, $k = 1, \ldots, K$, from the input into K feature spaces $(\Phi_1(\mathbf{x}_i), \ldots, \Phi_K(\mathbf{x}_i))$ each of which corresponding to a different kernel. Here D_k denotes the dimensionality of the k-th feature space. Now the aim is to learn a weighting over the different kernels. To do so one solves the following optimization problem (Bach et al., 2004),[11] which is equivalent to the linear SVM for $K = 1$:[12]

MKL Primal for Classification

$$\begin{aligned} \min \quad & \frac{1}{2}\left(\sum_{k=1}^{K} \|\mathbf{w}_k\|_2\right)^2 + C\sum_{i=1}^{N} \xi_i \\ \text{w.r.t.} \quad & \mathbf{w}_k \in \mathbb{R}^{D_k}, \boldsymbol{\xi} \in \mathbb{R}^N, b \in \mathbb{R}, \\ \text{s.t.} \quad & \xi_i \geq 0 \text{ and } y_i\left(\sum_{k=1}^{K} \langle \mathbf{w}_k, \Phi_k(\mathbf{x}_i)\rangle + b\right) \geq 1 - \xi_i, \;\; \forall i = 1, \ldots, N \end{aligned} \tag{4.7}$$

Note that (4.7) is different to taking a normal SVM and adding the kernels together as the regularizer $(\sum_k \|\mathbf{w}_k\|_2)^2$ instead of $(\sum_k \|w_k\|_2^2)$ is used, leading to a feature selection over kernels.

4.6.1.1 Solving the MKL Problem

In (Sonnenburg et al., 2006b) we proposed to reformulate this problem as a Semi-Infinite Linear Program (SILP), which can be derived using the dual formulation of (4.7) as suggested by Bach et al. (2004). The SILP formulation (4.8) can be solved using so-called *Column Generation* techniques (Hettich and Kortanek, 1993; Demiriz et al., 2002; Rätsch et al., 2002). The basic idea is to compute the optimal $(\boldsymbol{\beta}, \theta)$ for a restricted subset of constraints on $\boldsymbol{\alpha}$ of a Semi-Infinite Linear Program (SILP):

Semi-Infinite Linear Program

11. See (Sonnenburg et al., 2006a) for generalizations to other problem settings such as regression. Also note that very similar MKL algorithms can be found in (Weston, 1999; Bennett et al., 2002; Bi et al., 2004).
12. We assume $\mathrm{tr}(K_k) = 1$, $k = 1, \ldots, K$ and set d_j in (Bach et al., 2004) to one.

$$\begin{aligned} \max \quad & \theta & (4.8)\\ \text{w.r.t.} \quad & \theta \in \mathbb{R}, \boldsymbol{\beta} \in \mathbb{R}^K \\ \text{s.t.} \quad & \mathbf{0} \leq \boldsymbol{\beta}, \ \sum_k \beta_k = 1 \text{ and} \\ & \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y_i y_j \sum_{k=1}^{K} \beta_k k_k(\mathbf{x}_i, \mathbf{x}_j) - \sum_{i=1}^{N} \alpha_i \geq \theta & (4.9)\\ & \text{for all } \boldsymbol{\alpha} \in \mathbb{R}^N \text{ with } \mathbf{0} \leq \boldsymbol{\alpha} \leq C\mathbf{1} \text{ and } \sum_i y_i \alpha_i = 0 \end{aligned}$$

Then a second algorithm generates a new, yet unsatisfied constraint determined by $\boldsymbol{\alpha}$. These two algorithms iterate until guaranteed convergence (Hettich and Kortanek, 1993). It turns out that to find the most violated constraint, one needs to solve a single-kernel SVM problem using the intermediate solution $\boldsymbol{\beta}$ (Sonnenburg et al., 2005a). Using this idea we can thus take advantage of the efficient single-kernel SVM implementations (with and without `linadd` optimization).

A more efficient version (Sonnenburg et al., 2006b) adapts the $\boldsymbol{\beta}$'s while the chunking algorithm optimizes the $\boldsymbol{\alpha}$'s. This algorithm also requires the computation of linear combinations of kernels

$$g_i = \sum_{j=1}^{N} \alpha_j y_i \left(\sum_{k=1}^{K} \beta_k k_k(\mathbf{x}_i, \mathbf{x}_j) \right).$$

However, since the $\boldsymbol{\beta}$'s change during the optimization, one has to maintain iterates for every example *and* kernel:

$$g_{i,k} = \sum_{j=1}^{N} \alpha_j y_i k_k(\mathbf{x}_i, \mathbf{x}_j)$$

in order to compute $g_i = \sum_{k=1}^{K} \beta_k g_{i,k}$. Unfortunately, this approach is inefficient in combination with the common kernel caching strategies, since one now requires independent caching of K kernels, which considerably reduces the effectiveness when using large numbers of kernels or examples. Here, the `linadd` approach completely avoids kernel caching and can be straightforwardly applied to MKL. It turns out to be particularly effective as will be illustrated in simulation experiments in the next subsection. More details on these concepts and algorithms can be found in (Sonnenburg et al., 2006b).

4.6.1.2 Benchmarking MKL

The WD kernel of degree 20 consists of a weighted sum of 20 sub-kernels each counting matching d-mers, for $d = 1, \ldots, 20$. Using MKL we learned the weighting on the splice site recognition task for one million examples. Focusing on a speed comparison we now show the obtained training times for the different MKL algorithms

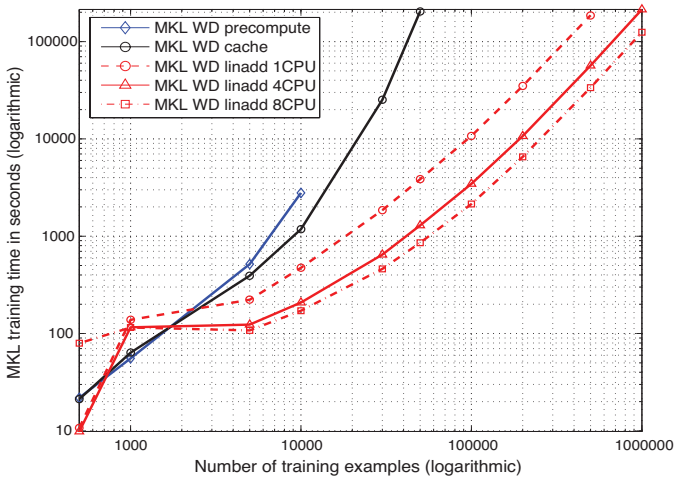


Figure 4.7: Comparison of the running time of the different MKL algorithms when used with the weighted degree kernel. Note that as this is a log-log plot, small appearing distances are large for larger N and that each slope corresponds to a different exponent.

applied to learning weightings of the WD kernel on the splice site classification task. To do so, several MKL-SVMs were trained using precomputed kernel matrices (*PreMKL*), kernel matrices which are computed on the fly employing kernel caching (*MKL*), MKL using the `linadd` extension (*LinMKL1*) and `linadd` with its parallel implementation[13] (*LinMKL4* and *LinMKL8* - on 4 and 8 CPUs). In contrast to the previous experiments, the SVMs regularization parameter was set to $C = 5$ and subproblem size was fixed at $Q = 112$. The results[14] are displayed in Table 4.7 and in Figure 4.7. While precomputing kernel matrices seems beneficial, it cannot be applied to large scale cases (e.g. $> 10,000$ examples) due to the $\mathcal{O}(KN^2)$ memory requirement for storing the kernel matrices.[15] On-the-fly-computation of the kernel matrices is computationally extremely demanding, but since kernel caching[16] is used, it is still possible on 50,000 examples in about 57 hours. Note that no WD-kernel specific optimizations are involved here, so one expects a similar result for arbitrary kernels.

The `linadd` variants outperform the other algorithms by far (speedup factor 53 on 50,000 examples) and are still applicable to datasets of size up to one million.

13. Algorithm 4.3 with the `linadd` extensions including parallelization of Algorithm 4.3.
14. Erratum: A programming error caused the optimizer to terminate the MKL SVM optimization before reaching the desired accuracy $\epsilon_{SVM} = 10^{-5}$. Since this affects the `linadd` *and* vanilla formulations, the benchmark comparison is still fair.
15. Using 20 kernels on 10,000 examples requires already 7.5GB, on 30,000 examples 67GB would be required (both using single precision floats)
16. Each kernel has a cache of 1GB.

Note that without parallelization MKL on one million examples would take more than a week, compared with 2.5 (2) days in the quad-CPU (eight-CPU) version. The parallel versions outperform the single processor version from the start achieving a speedup for 10,000 examples of 2.27 (2.75), quickly reaching a plateau at a speedup factor of 2.98 (4.49) at a level of 50,000 examples and approaching a speedup factor of 3.28 (5.53) on 500,000 examples (efficiency: 82% (69%)). Note that the performance gain using 8 CPUs is relatively small as for instance solving the QP and constructing the tries is not parallelized.

N	*PreMKL*	*MKL*	*LinMKL1*	*LinMKL4*	*LinMKL8*
500	22	22	11	10	80
1,000	56	64	139	116	116
5,000	518	393	223	124	108
10,000	2,786	1,181	474	209	172
30,000	-	25,227	1,853	648	462
50,000	-	204,492	3,849	1292	857
100,000	-	-	10,745	3,456	2,145
200,000	-	-	34,933	10,677	6,540
500,000	-	-	185,886	56,614	33,625
1,000,000	-	-	-	214,021	124,691

Table 4.7: Speed comparison when determining the WD kernel weight by Multiple Kernel Learning using the chunking algorithm (MKL) and MKL in conjunction with the (parallelized) `linadd` algorithm using 1, 4, and 8 processors (*LinMKL1*, *LinMKL4*, *LinMKL8*). The first column shows the sample size N of the data set used in SVM training while the following columns display the time (measured in seconds) needed in the training phase.

4.6.1.3 MKL Applications

Multiple Kernel Learning can be applied to knowledge discovery tasks. It can be used for automated model selection and to interpret the learned model (Rätsch et al., 2006; Sonnenburg et al., 2006b). MKL has been successfully used on real-world datasets in the field of computational biology (Lanckriet et al., 2004; Sonnenburg et al., 2005a). It was shown to improve classification performance on the task of ribosomal and membrane protein prediction (Lanckriet et al., 2004), where a weighting over different kernels each corresponding to a different feature set was learned. In their result, the included random channels obtained low kernel weights. However, as the data sets were rather small ($\approx 1,000$ examples) the kernel matrices could be precomputed and simultaneously kept in memory, which was not possible in (Sonnenburg et al., 2005a). There, we considered a splice site recognition task for the worm *C. elegans* and used MKL mainly to interpret the resulting classifier

(Sonnenburg et al., 2005b; Rätsch et al., 2006). In the next section we will propose alternative ways to facilitate understanding of the SVM classifier taking advantage of the discussed data representations.

4.6.2 Interpreting the SVM Classifier

One of the problems with kernel methods compared to probabilistic methods, such as position weight matrices or interpolated Markov models (Delcher et al., 1999), is that the resulting decision function is hard to interpret and, hence, difficult to use in order to extract relevant biological knowledge from it (see also Kuang et al., 2004; Zhang et al., 2003b). The resulting classifier can be written as a dot product between a $\boldsymbol{\alpha}$ weighted linear combination of support vectors mapped into the feature space that is often only implicitly defined via the kernel function:

$$f(\mathbf{x}) = \underbrace{\sum_{i=1}^{N_s} \alpha_i y_i \Phi(\mathbf{x}_i)}_{\mathbf{w}} \cdot \Phi(\mathbf{x}) = \sum_{i=1}^{N_s} \alpha_i y_i k(\mathbf{x}_i, \mathbf{x})$$

In the case of sparse feature spaces, as with string kernels, we have shown how one can represent $\mathbf{w}$ in an appropriate sparse form and then efficiently compute dot products between $\mathbf{w}$ and $\Phi(\mathbf{x})$ in order to speed up SVM training or testing. This sparse representation comes with the additional benefit of providing us with means to interpret the SVM classifier. For k-mer based string kernels like the spectrum kernel, each dimension w_u in $\mathbf{w}$ represents a weight assigned to that k-mer u. From the learned weighting one can thus easily identify the k-mers with highest absolute weight or above a given threshold τ: $\{\boldsymbol{u} \mid |w_u| > \tau\}$. Note that the total number of k-mers appearing in the support vectors is bounded by dN_sL where L is the maximum length of the sequences $L = \max_{i=1,\ldots,N_s} l_{x_i}$. This approach also works for the WD kernel (with and without shifts). Here a weight is assigned to each k-mer with $1 \leq k \leq d$ at each position in the sequence. While this approach will identify the k-mers contributing most to class discrimination, it is unsuitable for visualization of all possible $|\Sigma|^k$ substrings of length k per position. With growing order d, extracting all weights especially for the WD-kernel quickly becomes infeasible: the number grows exponentially in d ($\mathcal{O}(l|\Sigma|^d)$). Thus, one would need to accept a lower degree $\tilde{d} < d$ for visualization. However, this might lead for inferior generalization results (e.g. when using $\tilde{d}$ instead of d in training) or to an incomplete understanding of how the SVM is discriminating the sequences.

Extracting discriminative k-mers We therefore propose to choose a lower order $\tilde{d}$ just for visualization while making use of the *original* potentially higher order SVM classifier, by computing the contributions of the $k-$mers with $1 \leq k \leq d$ to the $\tilde{d}$-mers. The idea of this *k-mer extraction* technique is to identify all k-mers with $1 \leq k \leq d$ overlapping with the $\tilde{d}$-mers of the trained SVM classifier. The weights of the overlapping k-mers are then marginalized by the length of the match,

i.e. $w \mapsto \frac{1}{|\Sigma|^{l_p}} w$ where l_p is the length of the non-overlapping part. The marginalized weights are then summed up and assigned to the identified $\tilde{d}$-mers.

This can be done rather efficiently for the WD (shift) kernel: The sparse representation used for the WD kernels is a suffix trie at each position in the sequence. Thus all one needs to do is to traverse the tries – one after the other while adding up contributing weights. For example, if one wants to know the weight for the trimer `AAA` at position 42 for a WD kernel of degree 10, then 10-mers `XXXXXXXXXA` ($X \in \{A, C, G, T\}$) at position 33, 10-mers `XXXXXXAAAX` at position 36 and 10-mers `AXXXXXXXXX` at position 44, as well as all shorter overlapping $k-$mers ($k = 1 \dots 9$) contribute. Thus the weights in the contributing region `AAA` are collected, marginalized and added to the weight for `AAA` at position 42.

We have now obtained a weighting for $\tilde{d}-$mers for each position in the sequence: $W_{\boldsymbol{u},p}$, $p = 1 \dots l$, $\boldsymbol{u} \in \mathcal{U}$. Running the algorithm for different orders (e.g. $\tilde{d} = 1 \dots 8$), one may be interested in generating a graphical representation from which it is possible to judge where in the sequence which substring lengths are of importance. We suggest to compute this scoring by taking the maximum for each order per position,[17] i.e.

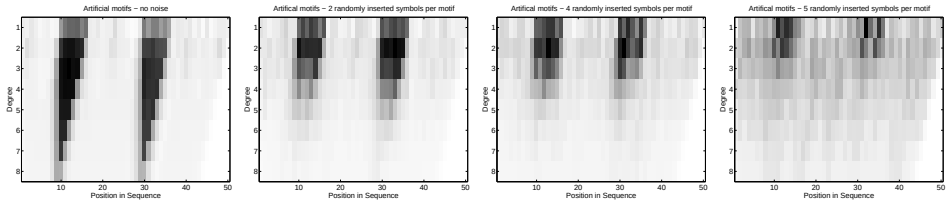
$$S_{\tilde{d},p} = \max(W_{\boldsymbol{u},p})_{\boldsymbol{u} \in \mathcal{U}}.$$

4.6.3 Illustration on toy data

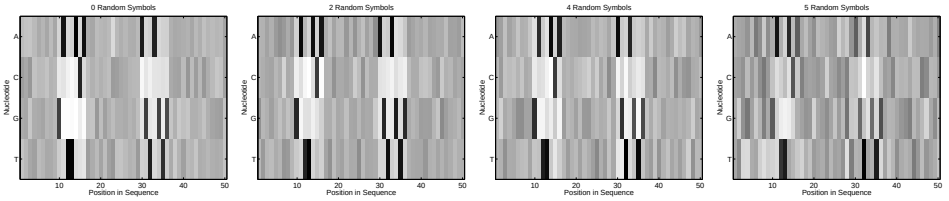
We will now use this approach for interpreting the SVM classifier. For a proof of concept, we apply the k-mer extraction technique to two toy data sets. For one dataset we insert two motifs at fixed positions in a random DNA sequence, which are to be detected by the Weighted Degree kernel. The other dataset is constructed in a way that it contains a small motif at a variable position. Here we use the Weighted Degree kernel *with shifts*.

Motifs at fixed positions For this toy dataset we generated $11,000$ sequences of length 50, where the symbols of the alphabet $\{A, C, G, T\}$ follow a uniform distribution. We choose $1,000$ of these sequences to be positive examples and hide two motifs of length seven: at position 10 and 30 the motifs `GATTACA` and `AGTAGTG`, respectively. To harden the problem we create different realizations of this dataset containing different amounts of noise: In the positive examples, we randomly replace $s \in \{0, 2, 4, 5\}$ symbols in each motif with a random letter. This leads to four different data sets which we randomly permute and split such that the first $1,000$ examples become training and the remaining $10,000$ validation examples. For every noise level, we train a SVM (parameters $C = 2$, $\epsilon = 0.001$) using the WD kernel of degree 20 followed by running the k-mer extraction technique. We also apply MKL, using a WD kernel of degree 7, to learn $M = 350$ WD kernel parameters (one

17. Other operators like for example the mean could also be considered.



(a) Weighting **S**: Weights assigned to each $\tilde{d}$-mer at each position in the sequence for different noise levels

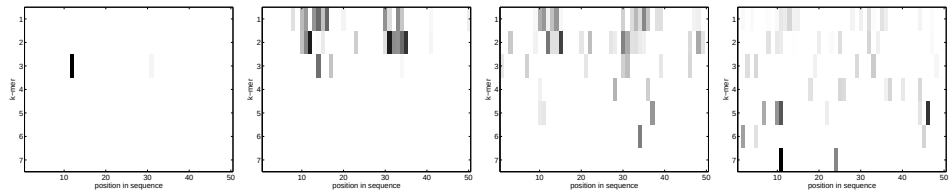


(b) Individual weights assigned to each 1-gram A,C,G,T at each position in the sequence for different noise levels

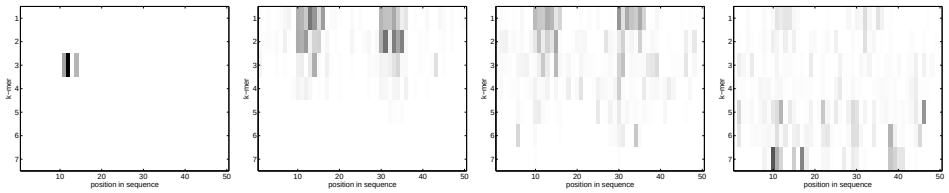
Figure 4.8: The figure illustrates positional weights **W** and **S** obtained using the k-mer extraction technique for increasing levels of noise (noise increases column-wise from left to right). Plots (a) shows the weighting assigned to d-mers from 1 to 8 for, where the x-axis corresponds to sequence positions and the y-axis to k-mer lengths. The plots (b) shows individual weights for 1-mers listed on the y-axis, where the x-axis corresponds to sequence positions. .

parameter per position and $k-$mer length). The results are shown in Figure 4.8 and 4.9. In the figures columns correspond to different noise levels – from no noise to 5 out of 7 nucleotides in the motifs being randomly replaced. For the WD-SVM (Figure 4.8), each figure in the upper row shows a scoring **S** where columns correspond to sequence position and rows to k-mer lengths used at that position. The lower row displays the k-mer weighting for $\tilde{d} = 1$, i.e. the weight assigned to each nucleotide A,C,G,T at each position in the sequence.

In Figure 4.9 (MKL), each figure shows the obtained kernel weights $\boldsymbol{\beta}$, where columns correspond to weights used at a certain sequence position and rows to the k-mer length used at that position. In contrast to the SVM, MKL leads to a sparse solution implied by the 1-norm constraint on the weighting $\boldsymbol{\beta}$. Hence, in the case of no noise one observes just a single 3-mer in the first motif to be sufficient to distinguish the sequences. The WD-SVM produces a rather dense solution. It exactly shows, however, that 7-mers at positions 10 and 30 have a large impact. With increasing noise level the motifs are split into smaller motifs. Thus one would expect shorter k-mers to be of importance. One indeed observes this tendency for the WD-SVM: The increased noise results in 4 and 3-mers to achieve the highest weights (2nd column). When 4 out of 7 nucleotides in each motif are randomly replaced, and 3 and 4-mers contribute most to discrimination. Considering the 1-gram weights with the highest scores at the positions 10-16 and 30-36 (sub-



(a) Single Trial MKL Weights



(b) Averaged MKL Weights

Figure 4.9: In this figures the columns correspond to the noise level, i.e. different numbers of nucleotides randomly substituted in the motif of the toy dataset. Each sub-plot shows the kernel weighting $\boldsymbol{\beta}$, where columns correspond to weights used at a certain sequence position and rows to the k-mer length used at that position. While the upper row displays a single trial $\boldsymbol{\beta}$, the lower row shows an averaged weighting obtained using 100 bootstrap runs, i.e. $1/100\boldsymbol{\beta}_{i=1}^{100}$

figure 4.8 (b)), the nucleotides compose the exact motifs embedded into the DNA sequences: `GATTACA` and `AGTAGTG`. Note that the classification performance also drops with increased noise from 100% auROC in the first two columns to 99.8% in the third and finally 85% when 5/7 nucleotides in each motif are noise. At that noise level the SVM picks up random motifs from regions not belonging to the hidden motifs. However the motifs – though a lot weaker – are still visible in the 1-gram plot. For Multiple Kernel Learning we have similar observations. However, when running the MKL algorithm on the data with 2 or more random nucleotides, it picks up a few unrelated k-mers. In (Sonnenburg et al., 2005a) we therefore proposed to use bootstrap replicates in order to obtain more reliable weightings (the average is shown in Figure 4.9 (b)) and additionally combined it with an statistical test such that only significant weights are detected. However, one can observe the same tendency as before: Shorter k-mers are picked up with increasing noise level. In column 3 a average ROC score of 99.6% is achieved. In the last column the ROC score drops down to 83%, while random k-mer lengths are detected.

Motif at varying positions In contrast to the previous paragraph, where we considered motifs at a fixed position, we are now study a dataset that contains a single motif somewhere in the interval $[49, 67]$. The dataset was generated in the following way: We created 5000 training and test examples which contain the letters C, G and T (uniformly, with the exception that C appears twice as often).

In the positive dataset we inserted the motif AAA randomly in the above interval

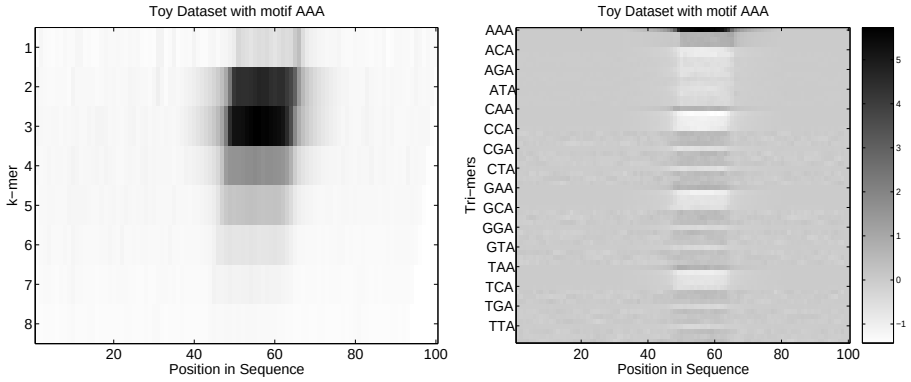


Figure 4.10: In the region $[49, 67]$ a motif AAA is hidden for the positive examples, and for the negative examples three A's are placed in the same region, but in most cases non-consecutive. This Figure shows the result of the k-mer extraction technique applied to an SVM classifier using the Weighted Degree Kernel *with shifts*. The left figure shows the $k-$mer importance per position (absolute values; darker colors correspond to higher importance). The right figure displays the weights assigned to each trimer at each position (gray values correspond to weights around zero and thus don't contribute in discrimination; white and light gray spots highlight silencers that add to a negative class label ; black and dark gray regions correspond to enhancers suggesting a positive class label).

and placed three A's (in most cases non-consecutive) in the same region for the negative examples. The task is again to learn a discrimination between the classes followed by a k-mer extraction analysis as above. While the Weighted Degree kernel *with shifts* is made for this kind of analysis it is obvious that the WD kernel would have a hard time in this setup, although due to the dominance of the AAA triplet in that region it will not fail totally. In this experiment we trained two SVMs one with the WD and one with the WDS kernel of order 20 and shift 15. We set SVM parameters to $\epsilon = 10^{-3}$ and $C = 1$. While the WD kernel achieves a respectable auROC of 92.8%, it is outperformed by the WDS kernel getting 96.6%. Running the k-mer extraction technique on the WDS-SVM result we obtain Figure 4.10. One clearly observes that 3-mers are getting the highest weights and one is able to even identify the motif AAA to be of importance in the range 49 to 64.

Real-World Splice Dataset We finally apply the same procedure to a splice dataset of (Sonnenburg et al., 2005a). This dataset contains 262,421 DNA sequences of length 141 nucleotides and was extracted by taking windows around a *C.elegans* acceptor splice site. We used a WD kernel of degree 20, trained a SVM ($\epsilon = 10^{-3}$ and $C = 1$) on the first 100,000 examples and obtained a classification performance

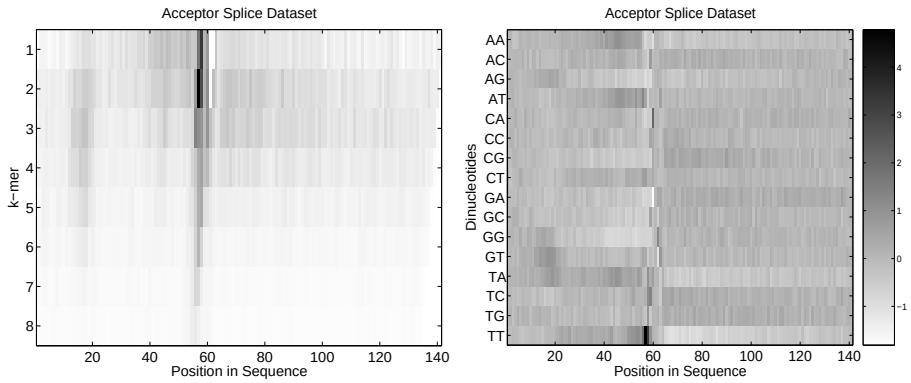


Figure 4.11: Results using the k-mer extraction technique on a splice dataset. The left figure shows the $k-$mer importance per position (absolute values; darker colors correspond to higher importance). The right figure displays the weights assigned to each trimer at each position (gray values correspond to weights around zero and thus don't contribute in discrimination; white and light gray spots highlight silencers that add to a negative class label ; black and dark gray regions correspond to enhancers suggesting a positive class label).

on the remaining 162,421 examples of auROC 99.7%. We then applied the k-mer extraction technique to the WD-SVM. The obtained results are shown in Figure 4.11. One observes a very focused signal in front of the acceptor splice site which is located between positions 60-61 followed by the `AG` consensus at positions 61 and 62 (see Sonnenburg et al., 2005a, for more details). It is also interesting to note that hexamers and pentamers are most discriminative at positions 57 and 58. Also a weak signal can be found around position 17 (33nt upstream of the splice site) which coincides with a upstream donor splice site as introns in *C.elegans* are often very short. Looking at the dinucleotide weighting one can even see an increased weighting for the `GT` consensus of the upstream donor splice. One also recognizes the known `T`-rich region in front of the splice sites as well as a strong aversion against `T`'s downstream of the splice site.

Discussion The presented k-mer extraction technique proves very useful in understanding the SVM classifier. It is advantageous compared to MKL as one can directly use the SVM performing best on a certain task. One does not have to retrain a MKL-SVM on a smaller dataset (MKL is computationally much more demanding) and also bootstrapping is not necessary. It will not only tell which degree $\tilde{d}$ is of importance and where in the sequence, but also highlights the locations of the exact motifs contributing most in discrimination. MKL will also work nicely to extract at which position in the sequence which k-mer length is important (at least when a additional statistical test is used). It is, however, not limited to kernels whose feature space is sparse and can be enumerated, but will also work with e.g.

RBF kernels as was shown in (Sonnenburg et al., 2006b). MKL is superior when one seeks a sparse solution and the data contains little noise. It might be necessary to incorporate the learned $\boldsymbol{\alpha}$ into the scoring.

4.7 Conclusion

This chapter proposes performance enhancements to make large scale learning with string kernels and any kernel that can be written as an inner product of sparse feature vectors, practical. The `linadd` algorithm (Algorithm 4.2), does not only speed up standalone SVM training but also helps to drastically reduce training times for Multiple Kernel Learning. Also, the sparse representation of the SVM normal vector allows one to look inside of the resulting SVM classifier, as each substring is assigned a weight. In a speed benchmark comparison the `linadd` algorithm greatly accelerates SVM training. For the standalone SVM using the spectrum kernel it achieves speedups of factor 60 (for the weighted degree kernel, about 7). For MKL we gained a speedup of factor 53. Finally we proposed a parallel version of the `linadd` algorithm running on a 8 CPU multiprocessor system which lead to *additional* speedups of factor up to 5.5 for MKL, and up to 4.9 for vanilla SVM training.

Acknowledgments The authors gratefully acknowledge partial support from the PASCAL Network of Excellence (EU #506778), DFG grants (JA 379/13-2, MU 987/2-1) and the BMBF project MIND (FKZ 01-SC40A). We thank Alexander Zien, Bernhard Schölkopf, Olivier Chapelle, Pavel Laskov, Cheng Soon Ong, Jason Weston and K.-R. Müller for great discussions. Additionally, we would like to express thanks to Alexander Zien for correcting and extending the visualization of contributions of k-mers for the WD-kernel.